Profile Point Component (PFP)

A Real-Time Profiling Component

Overview

- PFP stands for ***"profile point"***
- The component allows user to define a part of code that can be profiled for execution speed.
- PFP requires ***SYS/BIOS*** to be used. It makes use of ***BIOS hooks***.
- ***Profiling region*** is marked by a Begin-End function pair.
- During code execution the ***cycle count*** would be measured and stored inside the component internal storage.
- If the code was ***pre-empted*** during the execution of the profiling region, the measurement would be paused until the execution could resume.
- ***Nesting*** of profiling regions is allowed. Cycles measured in the outer regions will be ***exclusive***. They would only account for execution outside the inner regions.
- Current design and implementation have been tested only for measurements within the ***Tasks*** and ***Swi's***.

Code Example

- Initialization

```
...
pfpCreate();
pfpCalibrate(1000,0);

for (k = 0; k <= PFP_ID_LAST; k++) {
  pfpEnable(k); /* Enable PP #k */
}

pfpSetAlpha(PFP_ID_TASK0_1, 0.125);

BIOS_start();

...
```

- Usage

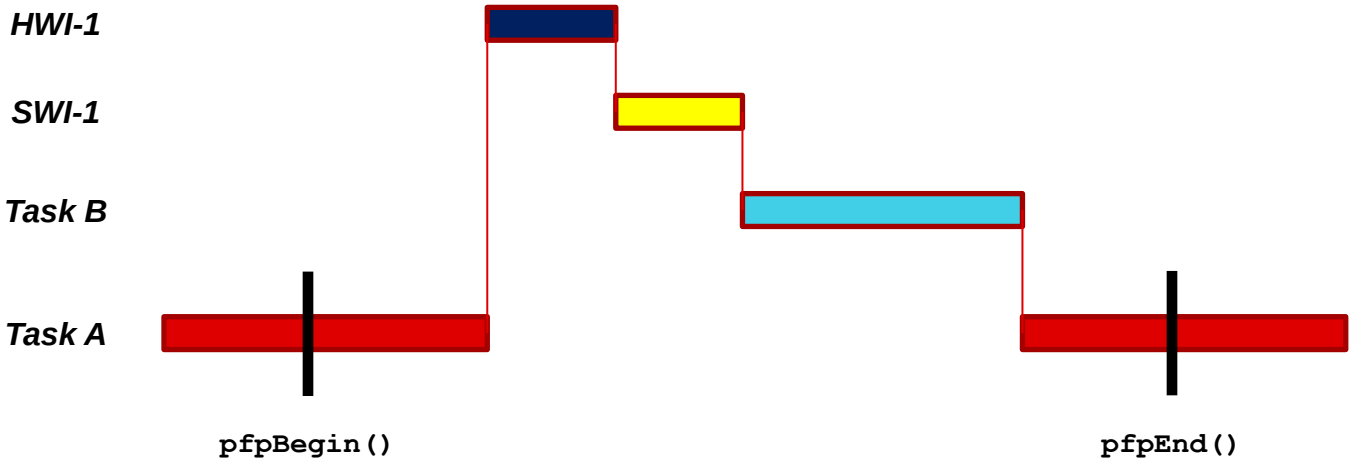
```
...
/* Start Measurement */
pfpBegin(PFP_TASK1_2, myHandle);

  doSomeWork(work);

pfpEnd(PFP_TASK1_2,0); /* Complete */

...
```

Execution Example



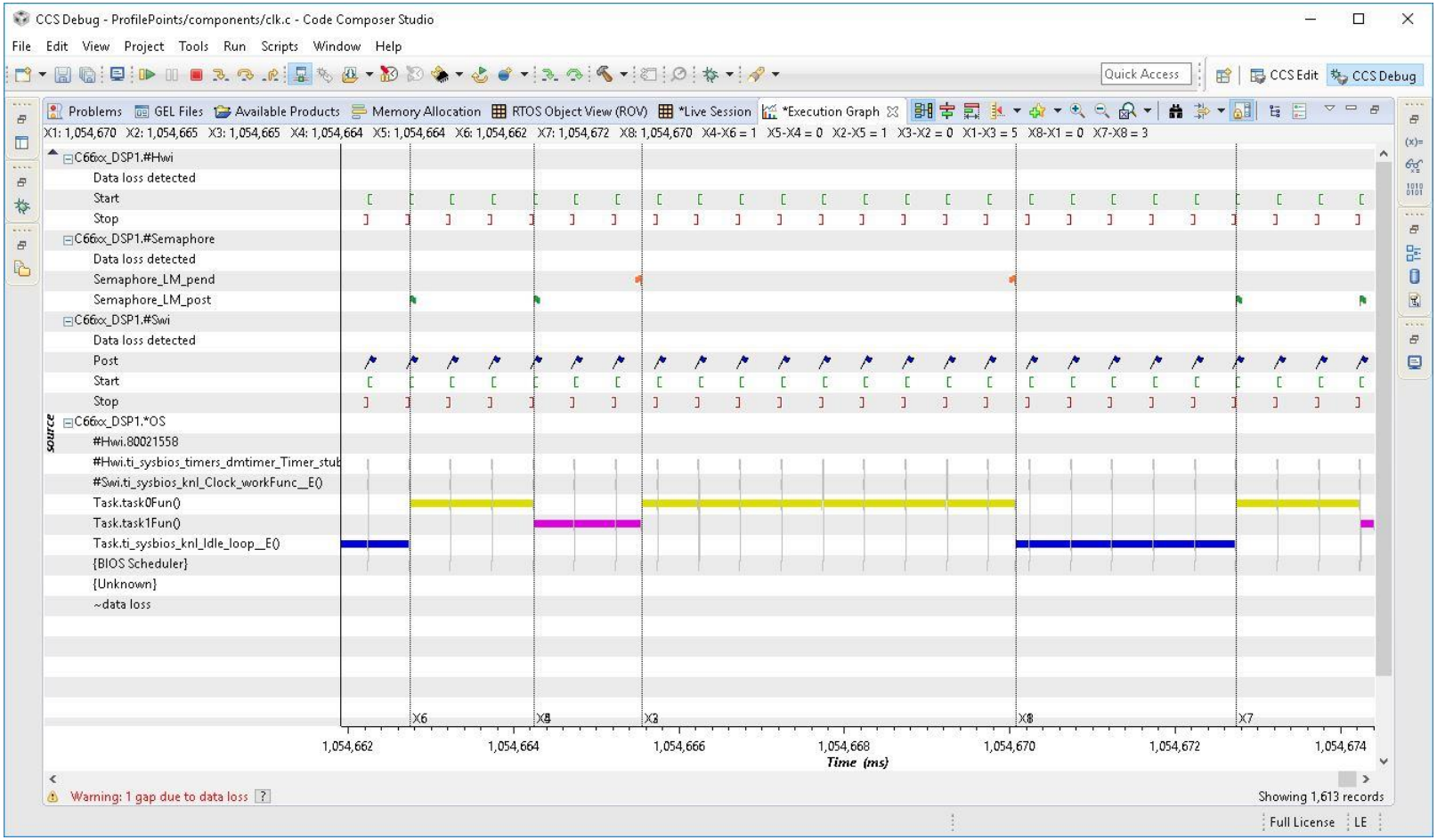
Measurement Example

```
ID: 00, n=1000, C=49265,     Cmin=49,      Cmax=314,     C-avg=49.265,      Avg-T=6.56867e-05ms
ID: 01, n=1000, C=750051197, Cmin=750048,  Cmax=752163,  C-avg=750051,      Avg-T=1.00007ms
ID: 02, n=100,  C=300473741, Cmin=3004727, Cmax=3005272, C-avg=3.00474e+06, Avg-T=4.00632ms
ID: 03, n=100,  C=150245906, Cmin=1502456, Cmax=1502491, C-avg=1.50246e+06, Avg-T=2.00328ms
ID: 04, n=100,  C=97626144,  Cmin=976259,  Cmax=976455,  C-avg=976261,      Avg-T=1.30168ms
```

- #0: Overhead measurement
- #1: Tuning of the dummy load to be close to 1ms
- #2: Task 0 inner 4ms dummy load
- #3: Task 0 outer 2ms dummy load + “other”
- #4: Task 1 dummy load

- Task 0: period 10ms, phase 0
- Task 1: period 10ms, phase 1.5ms

Execution Graph

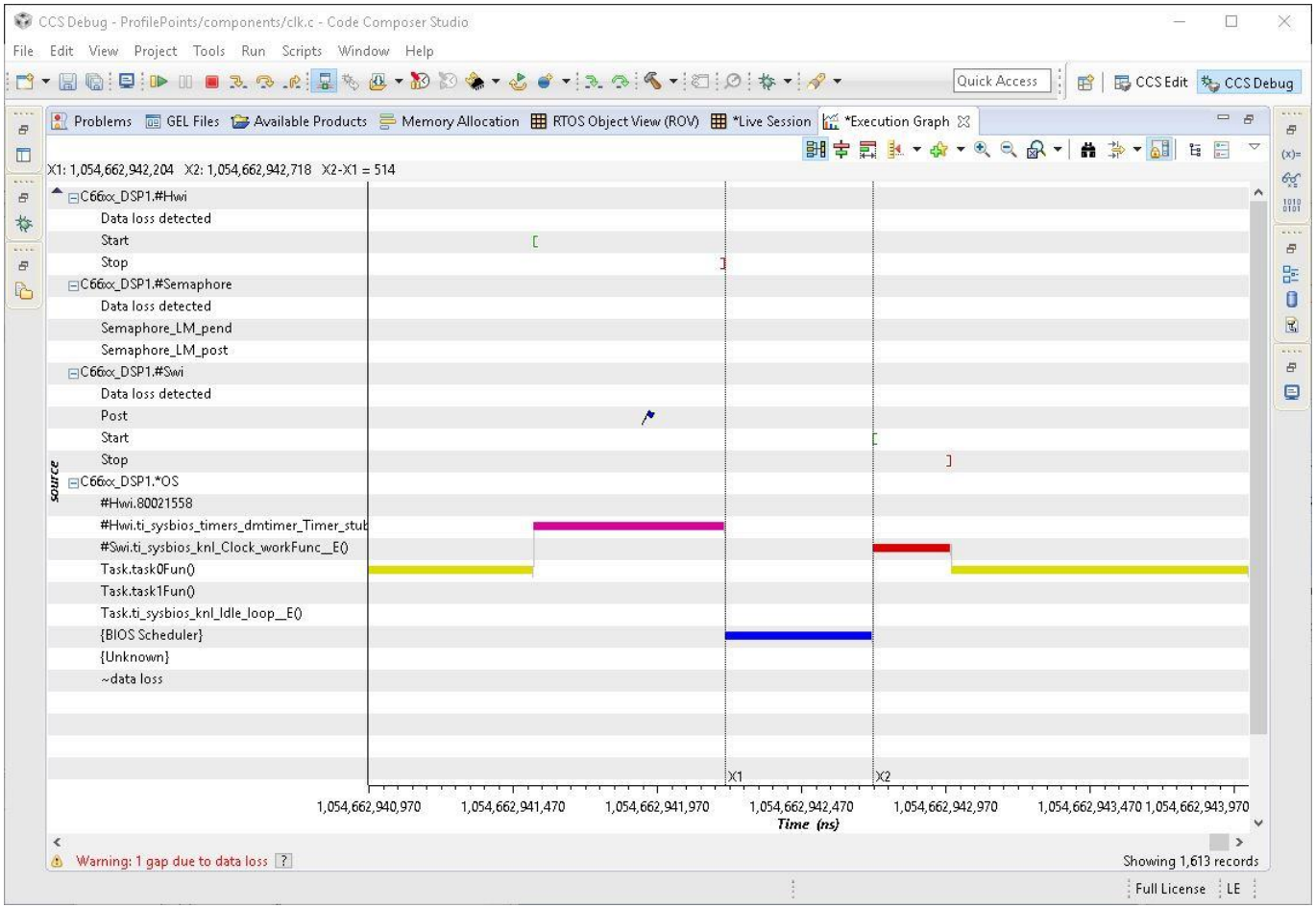


Measurements from Execution Graph

- Task 0: 6.026ms (vs. 6.0096ms PFP measured average)
- Task 1: 1.306ms (vs. 1.30168ms PFP measured average)

- Confirms that the measurements are done correctly.
- The Execution graph measurements include unwanted overhead from the clock Hwi/Swi
- PFP's advantages:
  - It automatically removes effects of task switching, pre-emption, and most of the overhead.
  - It allows easy measurement of any portion of task execution vs. whole task.
  - It allows for nesting of measurements.
  - It provide statistics and individual control of profile points.

Scheduling Overhead



API

- `void pfpCalibrate(int loopcnt, int reset)`
- `void pfpCreate(void)`
- `void pfpBegin(int id, void *exhandle)`
- `void pfpDisable(int id)`
- `void pfpEnable(int id)`
- `void pfpEnd(int id, int latch)`
- `void pfpGetStats(int id, pfpStats_t *pStats)`
- `void pfpResetStats(int id)`
- `void pfpSetAlpha(int id, float alpha)`

API – BIOS Hooks

- `void pfpTaskRegister(int hookSetId)`
- `void pfpTaskCreate(Task_Handle task, Error_Block *eb)`
- `void pfpTaskReady(Task_Handle task)`
- ***`void pfpTaskSwitch(Task_Handle prev, Task_Handle next)`***

- `void pfpSwiRegister(int hookSetId)`
- `void pfpSwiCreate(Swi_Handle swi, Error_Block *eb)`
- `void pfpSwiReady(Swi_Handle swi)`
- ***`void pfpSwiBegin(Swi_Handle swi)`***
- ***`void pfpSwiEnd(Swi_Handle swi)`***

- `void pfpHwiRegister(int hookSetId)`
- `void pfpHwiCreate(Hwi_Handle hwi, Error_Block *eb)`
- `void pfpHwiReady(Hwi_Handle hwi)`
- ***`void pfpHwiBegin(Hwi_Handle hwi)`***
- ***`void pfpHwiEnd(Hwi_Handle hwi)`***

Data Structures (PFP Descriptor)

Profile Point Descriptor (80 bytes on C66x)

```
struct pfpDescriptor_stc {
  void            *exhandle;    /* Execution object handle (hwi/swi/task) */
  int             id;           /* PP id which is the index into the pfpInst->pfpVector[] */
  int             scope;        /* BIOS_ThreadType */
  int             enable;       /* 1: enabled, 0: disabled */
  int             inside;       /* 1: inside (inside the measurement bracket), 0: outside */
  int             active;       /* 1: clock is active, 0: clock has been "paused" */
  int             latch;        /* 1: Latch the c_partial to spread over multiple PP regions */
  int             reset;        /* 1: reset stats in pfpBegin(), 0: do not reset */
  long long       c_total;      /* Total cycles (full count) */
  uint_least32_t  n_count;      /* Number of completed measurements */
  uint_least32_t  c_partial;    /* Current measurement */
  uint_least32_t  c_0;          /* Starting point time stamp */
  uint_least32_t  c_min;        /* Minimum measurement */
  uint_least32_t  c_max;        /* Maximum measurement */
  float           alpha;        /* Exponential averaging constant */
  float           c_average;    /* Exponentially averaged cycles */
  struct pfpDescriptor_stc *next;    /* Next element when in the active list */
  struct pfpDescriptor_stc *prev;    /* Previous element when in the active list */
};
typedef struct pfpDescriptor_stc pfpDescriptor_t;
```

Data Structures (PFP Context)

Profile Point Context ($24 + 80 \cdot N$ bytes on C66x, 32 PP's allocated at compile time)

```
struct pfpInst_stc {
  GateAll_Handle    ghandle;    /* Gate handle for Critical Section */
  pfpDescriptor_t   *head;      /* head of active profile points list */
  pfpDescriptor_t   *tail;      /* tail of active profile points list */
  uint_least32_t    overhead;   /* 0: if not calibrated */
  int               depth;      /* Current length of the active profile points list */
  int               active_id;  /* id of currently active PP */

  pfpDescriptor_t   pfpVector[PFP_PPCNT_MAX]; /* Statically allocated profile point vector */
};
typedef struct pfpInst_stc pfpInst_t;
```

- Profile point list contains the PP's that are currently with a measurement "in progress".
- The tail element is the most recent one and the one that could be currently active
- When pre-emption happens the currently active PP becomes inactive (clock paused)
- When PP is entered through `pfpBegin()` a new element would be put on the tail of the list and the element would become active (clock would start)
- When PP is exited through `pfpEnd()` the tail element is removed from the list and it is made to be inactive (clock stopped) and "outside". If measurement is not latched, statistics is updated at that point.

Dependencies

- PFP depends on:
  - SYS/BIOS 6.45.1.29
  - XDC Runtime 3.32.0.06 (time stamps, etc.)
- Data types:
  - Primarily C99
  - Relevant SYS/BIOS data types used as necessary
- Test Project uses:
  - CCS 6.1.3
  - UIA 2.0.3.43
  - AM572x EVM (one C66x core, 750MHz)

Notes on Implementation

- There is only minimal or no parameter checking within the PFP API
- Improper usage of functions may result in unexpected code behavior
- Every attempt was made to make measurements with the minimum overhead still preserving modular design and implementation
- **`pfpEnd()`** may be called prior to **`pfpBegin()`** when used inside loops
- **`pfpCreate()`** must be the first PFP function to call, usually in **`main()`**
- **`pfpCalibrate()`** (if used) must be called prior to first **`pfpBegin()`**
- **`pfpResetStats()`** and **`pfpDisable()`** should be called from the low enough execution priority in order to ensure the profile point we wish to reset or disable is not with a measurement in progress.
- PFP requires GateAll module of SYS/BIOS for Critical Sections
- Individual measurements must be below 5.7s duration (750MHz clock)
- Code was not tested if measurements are done inside HWI's

Future Work

- Overhead could be reduced if we would know where we were coming from when entering Hwi or Swi. Did we “pass” through a task or not?
  - If we were coming from a Task we should accumulate the time measured since the last capture of a timestamp. Otherwise we should not.
  - Problem is that we do not know where are we coming from when entering Hwi or Swi.
  - SYS/BIOS should provide a function that could return the “previously executing context” or handle. We could check it and find out if it was a task or not when entering Hwi or Swi.
- Testing should be done to check on usage within the HWI’s

PFP API Suggested Usage

- The following actions will be explained:
  - Initialization (creation, defaults, calibration, configuration & basic control)
  - Simple measurements
  - Getting and handling statistics
  - Latched measurements
  - Nested measurements
  - Loop measurements with task blocking
  - Invalid usage
  - Integration

PFP Component Initialization

- In order to use PFP component for profiling/benchmarking, one must initialize it first.
- The following actions need to be done (in the order shown):
  - **Creation:** see **`pfpCreate()`**. This must be the first thing to do.
  - **Calibration:** see **`pfpCalibrate()`**. This is *optional* and should be done after creation and prior to entering the first profile region. By default, measurement overhead of Begin-End pair is set to zero after the PFP component creation. Calibration uses PFP with ID #0 for the measurements.
  - **Configuration:** see **`pfpSetAlpha()`**. This is *optional* and it configures exponential averaging constant for a profile point. By default, exponential averaging is not configured for any of the profile points after the PFP component creation.
  - **Control:** see **`pfpEnable()`** and **`pfpDisable()`**. This enables or disables individual profile points. By default, all profile points are disabled after the PFP component creation.

Simple Measurements

- For simple cycle measurements see **`pfpBegin()`** and **`pfpEnd()`**
- Example:

```
...
Task_Handle myHandle;
myHandle = Task_self();
...
pfpBegin(PFP_ID_TASK0_1, myHandle);
   doSomeWork(work);
pfpEnd(PFP_ID_TASK0_1, 0);
```

Description:

- It is very important that the Begin-End pair are properly matched with the same PFP ID. For that reason one should use symbolic constants.
- The effect of this example is that the selected profile point would be used to measure the number of cycles consumed by the **`doSomeWork()`** function.
- The same profile point (**`PFP_ID_TASK0_1`**) should not be used for any other purpose in order to obtain meaningful execution statistics.
- If the code inside the Begin-End pair makes blocking calls, the time spent not running would not be included in the measurement.
- Begin-End PFP functions use SYS/BIOS GateAll module to synchronize access to internal PFP data

Getting and Handling Statistics

- For getting and handling PFP statistics see **`pfpGetStats()`** and **`pfpResetStats()`**.
- Example:

```
...
pfpStats_t pfp_stats;
...
pfpGetStats(PFP_ID_TASK0_1, &pfp_stats);
...
/* Somewhere else in the code... */
pfpResetStats(PFP_ID_TASK0_1);
```

Description:

- The statistics for the profile point indicated by **`PFP_ID_TASK0_1`** ID will be returned within the structure **`pfp_stats`**.
- The values returned are:
  - **`c_total`**: total number of cycles consumed so far
  - **`n_count`**: number of completed measurements
  - **`c_min`**: minimum number of cycles
  - **`c_max`**: maximum number of cycles
  - **`alpha`**: exponential averaging constant
  - **`c_average`**: exponentially averaged number of cycles
- Get-Reset statistics PFP functions use SYS/BIOS GateAll module to synchronize access to internal PFP data

Latched Measurements

- Measurements may be latched when calling **`pfpEnd()`** function
- Example:

```
...
pfpBegin(PFP_ID_TASK1_2, myHandle);
  doSomeWork(work1);
pfpEnd(PFP_ID_TASK1_2, 1); // latch stats
...
pfpBegin(PFP_ID_TASK1_2, myHandle);
  doSomeWork(work2);
pfpEnd(PFP_ID_TASK1_2, 0); // record stats
```

Description:

- The statistics for the profile point indicated by **`PFP_ID_TASK1_2`** ID will be calculated in two parts.
- The first part will be for "**`work1`**"
- The second part will be for "**`work2`**"
- The two measurements shown will be treated as a single one having work equivalent to "**`work1+work2`**".
- Latching can be done with any number of partial measurements. Only one measurement should be used to stop latching and record the stats.

Nested Measurements

- Nesting of Begin-End pairs is allowed. The outer profile regions will measure exclusive cycles.
- Example:

```
...
pfpBegin(PFP_ID_TASK0_2, myHandle);
  doSomeWork(work1);
  ...
  pfpBegin(PFP_ID_TASK0_1, myHandle);
    doSomeWork(work2);
  pfpEnd(PFP_ID_TASK0_1, 0);
  ...
pfpEnd(PFP_ID_TASK0_2, 0);
```

Description:

- The statistics for the profile point indicated by **`PFP_ID_TASK0_1`** ID will provide the estimate of "**`work2`**" load.
- The statistics for the profile point indicated by **`PFP_ID_TASK0_2`** ID will not include the "**`work2`**" load.

Loop Measurements with Task Blocking

- It is possible to measure task loop performance that includes blocking calls.
- Example:

```
...
loop {
  pfpEnd(PFP_ID_TASK0_2, 0);
  pfpBegin(PFP_ID_TASK0_2, myHandle);
  Semaphore_pend(semTask0WakeUp,
                 BIOS_WAIT_FOREVER);
  pfpBegin(PFP_ID_TASK0_1, myHandle);
    doSomeWork(work);
  pfpEnd(PFP_ID_TASK0_1, 0);
}
```

Description:

- In this case the profile point indicated by **`PFP_ID_TASK0_1`** ID will provide the estimate of "**`work`**" load.
- The statistics for the profile point indicated by **`PFP_ID_TASK0_2`** ID will not include the "**`work`**" load but would measure all other work done inside the loop (mostly overhead in this particular case).
- The time spent in a blocked state waiting for the semaphore to be posted would not be included in the measurement.

(In)valid Usage

- Every Begin call must have matching End call. (with the same PP ID)
  - If you do Begin-Begin-End with the same PP ID, the profile point will be disabled automatically due to invalid usage
- If you latch the measurement for certain PP, make sure you get to execute the final Begin-End pair that would update final statistics for a measurement.
- If you use exponential averaging, make sure you configured the averaging constant α properly to obtain the load estimates with correct time constant.
  - Use the following formula:

$$\alpha = \frac{T}{\tau}$$

    where T is the period with which the profile region would execute (or period within which you need to assess the load), and τ is the time constant.
  - For example, for period of 100ms and time constant of 500ms one should use 0.2 for the averaging constant.

Integration

- Code: **`pfp.h/.c`** – should be compiled using maximum optimization level
- GateAll BIOS module must be added to the configuration
- BIOS Hooks must be manually added to the configuration file if the current configuration tool does not support it
- Example config file is provided with the PFP test application
- pfpids.h or similar header file should be created and used for defining unique PFP ID's so that errors in usage would be avoided. Example is provided with the PFP test application.
- Follow instructions in this document regarding proper initialization and usage

PFP API Reference

- Please refer to the source code in **`pfp.h/.c`**